C/R Support for Heterogeneous HPC Applications

Konstantinos Parasyris, Leonardo Bautista Gomez
Barcelona Supercomputing Center (BSC)
E-mail: {konstantinos.parasyris,leonardo.bautista }@bsc.es

Keywords—Fault Tolerance, High-performance computing, Reliability

I. Extended Abstract

As we approach the era of exa-scale computing, fault tolerance is of growing importance. The increasing number of cores as well as the increased complexity of modern heterogenous systems result to substantial decrease of the expected mean time between failures. Among the different fault tolerance techniques, checkpoint/restart it is vastly adopted in supercomputing systems. Although, many supercomputers in the TOP 500 list use GPUs, only a few checkpoint restart mechanism support GPUs.

In this paper, we extend an application level checkpoint library, called fault tolerance interface (FTI), to transparently support multi-node/multi-GPU checkpoints. Upon a checkpoint invocation the extended library tracks the actual location of the data to be stored and handles the data accordingly. When data are stored in the GPU side, to hide the extra latencies, we overlap the copying of device memory to host memory with the writing of the data to the checkpoint file.

II. Importance of Fault Tolerance

The last decades the supercomputers have increased in size and computing cababilities. Exascale computing is the next objective, which will bring even more computing power to scientific applications and to industries. However several challenges are raised with exascale computing, the most important ones are the power consumption and the error resiliency. As the number of components increase in large scale systems, the systems become more error prone, and thus more prone to failures. It is expected that the next generation of high performance computing machines will experience failures up to several times an hour, making the need for effective fault resilience effective for building tomorrows HPC systems [1].

Another important consideration for the fault resiliency of extreme-scale HPC systems is the increasingly heterogenity of the components within the system. The future exascale will consit of multiple nodes with each node consisting of a high-performance system that combines a balance of high throughput general-purpose (GPGPU) pipelines for extreme high performance, coupled with high performance multicore CPUs targeting single-thread performance. The GPUs provide the high throughput required for exascale levels of computation, whereas the CPU cores handle hard-to-parallelize code sections and provide support for legacy applications. However, GPUs are more error prone than CPUs. In *TSUBAME* 40% of the total number of failures are caused by GPU errors, however the number of CPU related failures is below 5% [2]. When injecting faults [3] to a applications executing on CPUs, only 2.3% of the injected faults manifest as silent data corruptions, in GPUs this percentage rises to 16-33%. For all these reason the mean time between failures (MTBF) is expected to decrease even more in future systems.

To overcome failures, supercomputers typically use checkpoint restart techniques, by storing the state of the computation in reliable storage. Upon a failure, the most recent state is used to restart the computation. Unfortunately, the amount of data to be checkpoint increase, since HPC applications nowadays are able to process more information. On the one hand, the decrease of the MTBF results to higher checkpoint frequency to reduce the amount of re-computation. On the other hand, the increase of the data to store, increase the overhead of the checkpoint procedure. To make things even worse, typically, in GP-GPU HPC applications portions of the application data are stored in the CPU-memory, whereas other portions are stored in the GPU main memory. This distribution of data increases the overhead of the checkpoint procedure. For all these reasons, checkpoints reduce heavily the system's efficiency. To maintain high productivity in supercomputers and large data centers, it is important to: i) reduce the programmers effort to implement checkpoints ii) reduce as much as possible the data to be stored iii) reduce the total overhead of the checkpointing procedure.

We have extended a checkpoint library called Fault Tolerance Interface (FTI) [4] to support transparent checkpoint of data stored in differen CUDA-enabled GPU devices. Our method does not extend the library's API, but automatically tracks the physical memory location of user defined virtual addresses. The functionality transparently handles CPU, GPU as well as unified memory addresses. The methodology supports all the checkpointing techniques of the library, namely incremental checkpointing, hash based differential checkpointing and normal checkpointing.

III. FTI GPU/CPU implementation

FTI handles checkpoints in three different phases. The first, called initialization-phase, corresponds to the initialization of the library and the definition of the protected memory regions. The second layer, (Checkpoint/Restore(C/R))-phase, corresponds to the actual C/R procedure, hence it moves the data from the device and host memory to the stable local storage device (SSD, NVMe) and vice versa in the case of the recovering. When the checkpoint-phase is terminated, the application resumes the normal execution, and the async-phase starts. In the async-phase the FTI managers start in the background encoding and transferring the checkpoint files to the respective checkpoint level.

To support Hybrid GPU/CPU support to FTI we extend

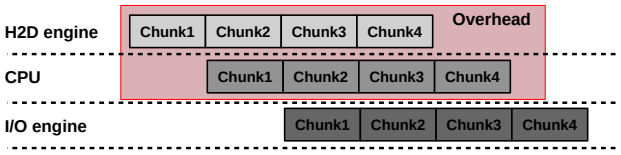

Fig. 1. Checkpoint communication scheme to overlap data transfers.

the initialization-phase and the C/R-phase. In the initialization-phase we identify the physical location of the address. The library through an FTI-call (*FTI_Protect*) identifies the physical location of the data. This is done through the CUDA driver API support, namely the function *cudaPointerGetAttributes(&attributes, address)*. The function raises an error when called with a host address, whereas it returns normally with a device or an UVA address. In the second case we further check the values of *attributes* field which provides information whether the address is UVA or not. In the end we tag each address as *CPU,GPU,MANAGED*.

During C/R phase, depending on the tag of each address we perform a different action. In the case of CPU or UVA addresses, we invoke the normal FTI C/R procedure. In the case of UVA addresses we use the CUDA driver to fetch the data from the GPU and move them to the stable local storage. Finally, in the case of *GPU* addresses, we overlap the writing of the file with the data movement from the GPU side to the CPU side. This is done through streams and asynchronous memory copies of chunks from GPU memory to host pinned memory. The procedure of transferring data from the GPU memory to the CPU is depicted in detail in Figure 1. Each protected memory region is divided into smaller blocks. The size of the block, from now on called communication block (*cBlock*), is controlled through a configuration option. The CPU requests from the Host to Device (H2D) engine an asynchronous transfer of the first *cBlock*, when the *cBlock* is copied to the host memory, the CPU requests the next *cBlock* and starts performing the necessary actions with the current cBlock. The main actions are the following: i) Update the checkpoint integrity checksum ii) Copy the *cBlock* to the I/O layer through the respective I/O library call.

When the data are copied to the I/O layer the CPU starts processing the next chunk, which ideally should already be copied in the host memory. The application process does not wait for the I/O operations to finalize, when all data are moved to the I/O layer it informs the FTI-managers to start the background actions and resumes the user code execution. The described scheme is optimal only if the execution time to compute the integrity checksum and copy the *cBlock* to the I/O layer is equal to the execution time needed to copy the data from the device to the host. In any other case, either the H2D engine or the CPU is idle.

A. Evaluation

In this section we analyze the FTI GPU checkpoint scheme. We use a micro-benchmark for profiling and analysis purposes. The micro-benchmark checks the strong/weak scaling of our approach using different mixtures of device/host memory allocations. The micro-benchmark allocates two memory buffers, the first buffer, called *hBuff*, is allocated on the host memory, whereas the second one, called *dBuff*, is allocated on the device

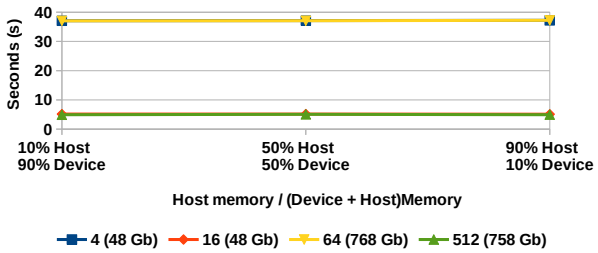

Fig. 2. Execution time of the checkpoint procedure for the configurations

memory. The size of each memory buffer is user defined. The application protects these two buffers and performs a checkpoint every 5 minutes. To test both the weak and the strong scaling of our approach we execute each benchmark on 4 different node/process mappings. Specifically, we executed experiments using 1 and 16 nodes, with 4 and 32 processes in each node. The checkpoint size of each node is 48 Gb regardless the number of processes. On each of these mappings we execute 3 different memory allocation schemes. In the first scheme, for each process we allocate 10% memory on the host and the remaining 90% to the device memory. The second scheme allocates 50-50% on the respective memories and the final scheme allocates 90-10%. In Figure 2 we depict the results of the executed experiments the X-axis represents the memory schemes, the Y-axis the amount of time spend by the user process to perform a single checkpoint and the different lines correspond to the different node/process mappings.

IV. Conclusions

Interestingly, regardless the actual memory location of the checkpoint data the checkpoint overhead remains the same. The implementation demonstrates nice weak and strong scaling. We profiled the execution time of the checkpoint procedure. The communication between the GPU and the CPU is completely overlapped, therefore there is almost no-overhead to move the data from the GPU to the CPU. Interestingly, the execution time is mainly spend in computing the integrity checksum of the checkpoint file. We plan to move the computation of the checksum on the GPU and overlap it with the actual writing of the C/R file. This will dramatically decrease the execution time of the checkpoint procedure.

References

[1] F. Cappello, "Fault tolerance in petascale/ exascale systems: Current knowledge, challenges and research opportunities," *The International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 212–226, 2009.

[2] B. Pourghassemi and A. Chandramowlishwaran, "cudacr: An in-kernel application-level checkpoint/restart scheme for cuda-enabled gpus," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2017, pp. 725–732.

[3] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer, "Hauberk: Lightweight silent data corruption error detector for gpgpu," in *2011 IEEE International Parallel Distributed Processing Symposium*, May 2011, pp. 287–300.

[4] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "Fti: High performance fault tolerance interface for hybrid systems," in *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2011, pp. 1–12.

Konstantinos Parasyris is a Postdoctoral researcher at the Barcelona Supercomputing Center where he works on Resilience and Optimization. He received the B.Sc., M.Sc., and Ph.D. at the electrical and computer engineering at the university of Thessaly (Greece). His research is focused on fault tolerance and application level error resiliency. During his PhD he developed GemFI a simulation based fault injection tool as well as XM2 a hardware level fault injection. He has been involved in FP7 ScoRPiO and H2020 Uniserver European projects. In ScoRPiO he actively participated in the development of a task-based approximate programming model, whereas in Uniserver he participated on dynamically identifying the voltage margins of modern processors for reliable operation.